Table of Contents:

5/11/2025 - 5/17/2025

Robotic Manufacturing Group GitHub For Project (Will Need Access):

https://github.com/mistone-eng/stem-manufacturing

5/13/2025

Initial Setup and Configuration

- The Raspberry Pi has been successfully set up and connected to a GitHub repository for efficient code management. This allows code to be managed through VS Code on a laptop, with the Raspberry Pi updating via a simple “git pull” command.
- The default microphone (Blue Yeti Stereo Microphone) has been set up as the default audio device for the Raspberry Pi.
- Initial success: Audio recording and playback through the microphone are functioning properly, confirming proper microphone setup.

Speech Recognition System Development

- The initial Python script for speech recognition (attached as Screenshot 1) was developed. This script is designed to take voice input and print the recognized speech in the terminal.
- An additional Python script was explored from two YouTube videos and GitHub repositories (links below), offering alternative methods:
    - GitHub Repository: AudasWasTaken/Voice_Recognition_RPI
    - YouTube Video 1: Voice Recognition on Raspberry Pi - Simple Setup
    - YouTube Video 2: Advanced Speech Recognition with API Options
- The voice recognition system is not yet functioning. Various errors have been encountered related to the SpeechRecognition package, and potential solutions using paid APIs have been noted in Video 2 (Don’t want to jump to this yet).

Screenshots:

```python
import speech_recognition as sr

def recognize_speech_from_mic():
    recognizer = sr.Recognizer()

    # Explicitly use the Yeti Microphone (Card 2)
    microphone = sr.Microphone(device_index=2)

    try:
        with microphone as source:
            print("Adjusting for ambient noise... Please be silent for a moment.")
            recognizer.adjust_for_ambient_noise(source, duration=1)
            print("Listening now... Speak clearly.")
            audio = recognizer.listen(source)

        print("Recognizing...")
        text = recognizer.recognize_google(audio)
        print("You said: " + text)

    except sr.RequestError:
        print("Could not request results from Google Speech Recognition service.")
    except sr.UnknownValueError:
        print("Sorry, I could not understand what you said.")
    except AssertionError as e:
        print(f"Microphone Error: {e}")

if __name__ == "__main__":
    print("Speech to Text - Microphone Input")
    recognize_speech_from_mic()
```

Screenshot: Initial Python Speech Recognition Script

```python
import speech_recognition as sr
from datetime import date
from gpiozero import LED
from time import sleep

red = LED(17)
relay1 = LED(14)
relay2 = LED(15)

r = sr.Recognizer()
mic = sr.Microphone()

print("hello")

while True:
    with mic as source:
        audio = r.listen(source)
    words = r.recognize_google(audio)
    print(words)

    if words == "today":
        print(date.today())

    if words == "LED on":
        red.on()

    if words == "LED off":
        red.off()

    if words == "relay one on":
        relay1.on()

    if words == "relay one off":
        relay1.off()

    if words == "relay two on":
        relay2.on()

    if words == "relay two off":
        relay2.off()

    if words == "exit":
        print("...")
        sleep(1)
        print("...")
        sleep(1)
        print("...")
        sleep(1)
        print("Goodbye")
        break
```

Screenshot: Alternative Python Script (YouTube/GitHub)

GPIO Control Issues and Debugging

- The Raspberry Pi has encountered consistent issues with GPIO control.
- Testing with various scripts and GPIO packages (RPi.GPIO, gpiozero, pigpio, lgpio) has consistently failed to activate GPIO pins.
- Testing included connecting an LED via a 220-ohm resistor, but GPIO output remains inactive.
- Troubleshooting steps attempted:
    - Testing in a virtual environment.
    - Trying different kernels.
    - Switching between different GPIO packages.
- This remains a critical issue to resolve in the upcoming weeks.

Upcoming Tasks and Objectives

- Resolve GPIO control issues to enable hardware control through Python scripts.
- Achieve a fully working speech recognition system where voice commands (e.g., "LED on") can control GPIO pins to activate the corresponding hardware.
- Explore alternative SpeechRecognition APIs or implement API keys for higher accuracy, as suggested by the YouTube video.
- Refine and test the Python speech recognition script until it can consistently recognize and execute voice commands.

References and Resources

- GitHub Repository: AudasWasTaken/Voice_Recognition_RPI
- YouTube Video 1: Voice Recognition on Raspberry Pi - Simple Setup
- YouTube Video 2: Advanced Speech Recognition with API Options

5/25/2025 - 5/31/2025

5/28/2025

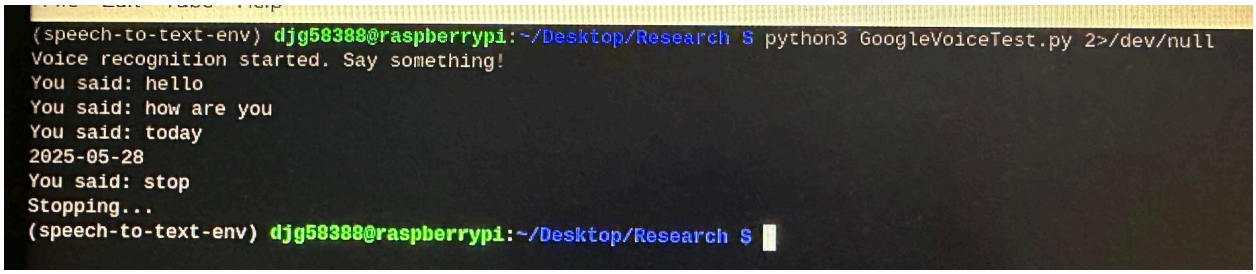
Successful Speech Recognition Execution

The speech recognition module has been successfully executed using a Python script named GoogleVoiceTest.py, fulfilling the primary task for the week. This script uses the Google Speech Recognition API to capture and transcribe voice commands in real-time. Key features include:

- Suppression of ALSA errors for cleaner terminal output.
- Ambient noise calibration.

- Recognition of keywords like **“**today**”** and **“**stop**”** with respective actions:
    - Prints the current date when “today” is said.
    - Exits the loop when “stop” is said.
- Error handling for unrecognized or failed speech attempts.

The script was verified to be functional using a Blue Yeti US.B microphone connected to the Raspberry Pi.



Screenshot: terminal output during successful voice recognition.

Preparation for Voice-Controlled DOBOT Integration

With the speech recognition pipeline operational, the next phase of development involves integrating it with the DOBOT robotic arm. This integration aims to enable voice-based control of robotic actions, allowing users to command the robot using intuitive phrases.

Next steps include:

- Research DOBOT Control APIs: Determine whether control will be handled via USB, TCP/IP, or a Python SDK.
- Test Command Structure: Identify a test case like “DOBOT, pick up gear” or “DOBOT, reset arm.
- Assess Safety Requirements: Consider how to pause or interrupt motion using voice.
- Sketch Initial Command Map:
    - “Start disassembly”
    - “Move left/right”
    - “Stop now”

Potential applications of this setup include:

- Hands-free interaction for disassembly tasks.
- Enhanced accessibility and ergonomics.
- Modular extensions for other arms or inspection tools.

References and Demo Resources

*The python and README files can be found in my GitFront linked at the top of this document.

- GoogleVoiceTest.py: Finalized Python script for voice capture and keyword detection.
- README.txt: Step-by-step instructions for environment activation and script execution.
- Video Demo: Functionality walk-through of the speech-to-text system in action.

5/29/2025

Dobot Magician Lite - Physical Setup and Fundamentals

With the speech recognition module verified as operational, this week's efforts shifted toward learning the fundamentals of the DOBOT Magician Lite and exploring how to integrate it with Python and voice commands. The goal is to enable full voice-activated robotic control for basic mechanical tasks using a USB-connected Raspberry Pi.

The DOBOT Magician Lite robotic arm is a 4-axis desktop robot designed for educational and light industrial tasks. The claw gripper (mechanical) will be used for object manipulation such as gear pickup.

Step-by-Step Setup:
1. Ensure Power is OFF before handing EoATs or USB cables

2. Attach the Claw Gripper (EoAT):

- Press and hold the EoAT release button under the arm
- Snap in the claw gripper until it locks securely
- (Another Option) Connect the vacuum tube (if present) for gripper operation

*Note: The claw gripper is servo-driven and does not require vacuum tube connections. Vacuum lines are only used for suction or soft grippers.

3. Connect the Robot:

- Plug the 12V DC power supply into the wall and robot
- Connect the USB cable from the DOBOT to the Raspberry Pi
- Wait for the DOBOT to fully power on (GREEN light and startup beep)

4. Initial Homing and Software Start:

The DOBOT must be homed to initialize its internal coordinate system before any movement commands are issued. Homing resets the arm to its origin positions, ensuring movement accuracy and repeatability.

There are two primary methods to perform the homing procedure:

Option A: Homing via Python SDK on Raspberry Pi (*Preferred)

- Install Required Libraries:

```
sudo apt update
sudo apt install python3-pip
pip3 install pyserial
```

Screenshot: Command to install libraries.

- Clone DOBOT Python Library:

```
git clone https://github.com/AlexGustafsson/dobot-python.git
cd dobot-python
```

Screenshot: Command to clone and direct to github repository.

- Identify the USB Port

```
ls /dev/ttyUSB*
# Example output: /dev/ttyUSB0
```

Screenshot: Command to identify USB Port.

- Write and Run Homing Script

  Create a new python script (eg. dobot_home.py)

```
from pydobot import Dobot

device = Dobot(port='/dev/ttyUSB0', verbose=True)

print("Homing DOBOT...")
device.home()
print("Homing complete.")

device.close()
```

Screenshot: Homing Python Script

Save and run the script ($ python3 dobot_home.py)

- The DOBOT will beep and return to the home position just as it would be the next option.

Option B: Homing with DobotLab (Graphical Interface)

- Install DobotLab Software: https://www.dobot.cc/downloadcenter/dobot-magician-lite.html
- Connect the Robot via USB
    - Plug the robot into your computer using the USB cable
    - Ensure the power is connected and the green indicator is active
- Launch DobotLab
    - Once launched, navigate to the “Teaching & Playback Lab”
- Establish Communication
    - Click the “Connect” button in the bottom-left corner
    - Select the correct COM port (eg. COM3) or device name and wait for status to show “Connected”
- Home the Robot
    - Open the Arm Control Panel and click the “Home” button
    - The arm will perform a homing sequence and settle in the neutral position
- (*Optional*): set the end effector to “claw gripper” under the Tool Settings for manual movement or testing.

This completes the physical and software setup required to prepare the DOBOT Magician Lite for Python-based control. All future movement commands and voice integrations will assume the arm has been homed prior to execution.

Python API Usage and Testing on Raspberry Pi

With the DOBOT Magician Lite successfully homed using the Python SDK, this section documents how to implement programmatic control of the robotic arm using Python scripts executed directly from the Raspberry Pi.

The control library used is dobot-python, a lightweight interface that communicates with the DOBOT via USB.

Environment Setup Recap *Note: These steps would have been completed if you completed Option A for homing.

- Installing control library:

```
sudo apt update
sudo apt install python3-pip
pip3 install pyserial
git clone https://github.com/AlexGustafsson/dobot-python.git
cd dobot-python
```

Screenshot: Commands to install control library

- Identifying the connected DOBOT port:

```
ls /dev/ttyUSB*
# Example result: /dev/ttyUSB0
```

Screenshot: Command to identify connected DOBOT port

Example Code: Basic Motion and Gripper Commands (No speech-recognition)

The following Python script serves as a test case for **verifying motion commands** and claw gripper activation:

```python
from pydobot import Dobot

device = Dobot(port='/dev/ttyUSB0', verbose=True)

# Always home before movements
device.home()

# Move to pickup location
device.move_to(x=200, y=0, z=30, r=0, wait=True)
device.grip(True)  # Close gripper

# Move to place location
device.move_to(x=150, y=100, z=30, r=0, wait=True)
device.grip(False)  # Open gripper

device.close()
```

Screenshot: Python script for basic arm movement and grip actions.

Key Methods to Note:

- device.move_to(x, y, z, r, wait=True) -> Cartesian motion
- device.grip(True/False) -> Close/open claw gripper
- device.home() -> **ALWAYS** call this before the first movement
- device.close() -> terminates serial communication safely

Additional Notes:

- Always move vertically away from the table (Z-axis up) before any horizontal repositioning
- Add delays or error handling when chaining multiple motions for precision

Voice Command Integration with DOBOT

The goal of this section is to create a new Python script that captures voice commands, interprets key phrases, and triggers corresponding robot actions using the Python API.

This functionality is achieved using the *speech_recognition* library and Google's built-in API.

Voice Recognition Setup

```
pip3 install SpeechRecognition
```

Screenshot: Command to install SpeechRecognition Package

No API key or billing is required when using the Google Web Speech API via speech_recognition. It supports short real-time queries and is sufficient for this project's scope.

Example Code: Voice-Controlled DOBOT script

Imports, variable assignments, and device setup:

```
import speech_recognition as sr
from pydobot import Dobot
import os, sys

# Suppress ALSA errors on Raspberry Pi
sys.stderr = open(os.devnull, 'w')
recognizer = sr.Recognizer()
mic = sr.Microphone()
sys.stderr = sys.__stderr__

# Connect to DOBOT
device = Dobot(port='/dev/ttyUSB0', verbose=True)
device.home()  # Homing at script start

print("Voice control initialized. Say a command.")
```

Voice Command Options:

```
while True:
    with mic as source:
        recognizer.adjust_for_ambient_noise(source)
        audio = recognizer.listen(source)

    try:
        command = recognizer.recognize_google(audio).lower()
        print(f"You said: {command}")

        if "pick" in command:
            device.move_to(x=200, y=0, z=30, r=0, wait=True)
            device.grip(True)

        elif "place left" in command:
            device.move_to(x=150, y=100, z=30, r=0, wait=True)
            device.grip(False)

        elif "place right" in command:
            device.move_to(x=250, y=-100, z=30, r=0, wait=True)
            device.grip(False)

        elif "reset" in command or "home" in command:
            device.home()

        elif "stop" in command:
            print("Stopping system.")
            break
```

Error Handling and Clean device closure

```
    except sr.UnknownValueError:
        print("Could not understand audio.")
    except sr.RequestError:
        print("Speech recognition service unavailable.")

device.close()
```

Recommended Voice Commands And Structure

Command Phrase	Action Performed
“Pick up” or “Pick”	Move to object and close claw
“Place left”	Move to drop zone (left) and release
“Place right”	Move to drop zone (right) and release
“Home” or “Reset”	Send robot to home position
“Stop”	Exit the control loop

I was thinking about creating user-defined functions for these command phrases so that the command would be something as simple as:
PlaceRight(Position), PlaceLeft(Position), etc.

Tip: Use *in* statements for flexible recognition and lowercase **ALL** input (*command.lower()*)

Troubleshooting Tips

- If no response from the DOBOT, confirm correct port with:

  $ ls /dev/ttyUSB*

- Make sure microphone input is functioning using:

```
with mic as source:
    audio = recognizer.listen(source)
print("Audio captured.")
```

- Consider using try/except blocks around robot motion to prevent crashes on invalid movement.

References

- https://github.com/SERLatBTH/StarterGuide-Dobot-Magician-with-Python
- https://github.com/AlexGustafsson/dobot-python/tree/master/examples
- https://www.dropbox.com/scl/fi/6heeobwcnfa85rtci6oj5/Copy-of-V1.1-Dobot-Lite-Curriculum-2024.pdf?rlkey=57hikoqjfkav8pdnc9shyk616&st=6lc6pew1&dl=0

DOBOT Python Function Development and Testing

Objective

Develop custom Python functions for the DOBOT Magician Lite to perform homing, resetting, and pick-and-place tasks using the pydobot GitHub repository. Each function was defined, tested with a dedicated script, and verified through physical execution using the Raspberry Pi. This lays the groundwork for full voice-command integration.

The GitHub Repository mentioned on 5/29/2025 proved useless during testing. **Due to this we moved on to the linked GitHub Repository**:
https://github.com/luismesas/pydobot/tree/master

We cloned this repository onto the raspberry pi and began creating some Python Functions that can be used when programming the DOBOT robotic arm. These functions will be described in detail below.

home() Function and Test

We began by creating a home() function within the dobot.py script. This function safely returns the robotic arm to a defined neutral “home” position, centered and elevated to avoid collision with the workstation.

```
def home(self):
    """
    Moves the robot to a predefined physical home position (centered and up).
    This is not mechanical homing — it is a soft reset.
    """
    # # Clear and start the queue to ensure clean state
    # self._set_queued_cmd_clear()
    # self._set_queued_cmd_start_exec()

    # Define safe center coordinates
    home_x = 200.0
    home_y = 0.0
    home_z = 70.0
    home_r = 0.0

    # Move to defined "home" pose
    self.move_to(home_x, home_y, home_z, home_r, wait=False)

    if self.verbose:
        print(f'pydobot: moved to soft home position at ({home_x}, {home_y}, {home_z}, {home_r})')
```

Screenshot: Defined home() function in dobot.py file.

To test this function, we developed a script called test_home.py, which follows this process:

1. Sends the robot to the defined home position to ensure a consistent starting point

2. Moves the arm to a random offset location (simulating work).

3. Calls home() again to validate that the arm reliably returns to its original home position

4. Closes the device

A screenshot of this test script can be seen below.

```python
from pydobot import Dobot
from serial.tools import list_ports
import time

# Step 1: Automatically detect the DOBOT's port
port = list_ports.comports()[0].device
device = Dobot(port=port, verbose=True)

# Step 2: First Home - should go to physical zero, then center
print("\n--- Homing to center (first time) ---")
device.home()
time.sleep(2)

# Step 3: Move to test position
print("\n--- Moving to test position ---")
device.move_to(x=250.0, y=0.0, z=50.0, r=0.0, wait=False)
time.sleep(2)

# Step 4: Second Home - should return to same center position
print("\n--- Homing to center (second time) ---")
device.home()
time.sleep(2)

# Step 5: Done
device.close()
print("\n--- Test complete ---")
```

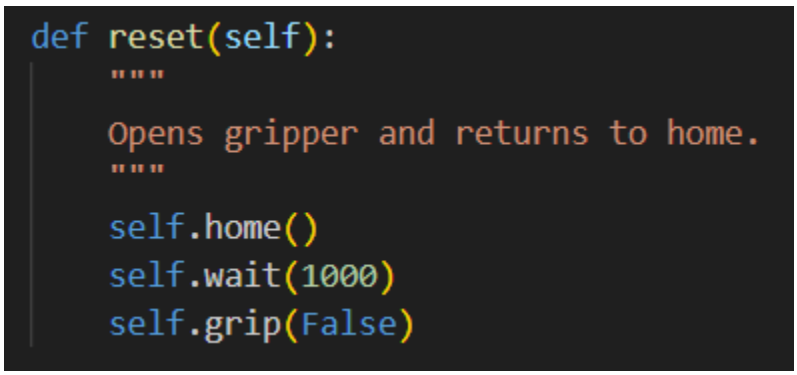
Screenshot: test_home.py - Test script for the homing command

Below is a link that shows the source code along with a video demonstration of the test script executing successfully:

https://drive.google.com/drive/folders/1BIL-ji7wjx0Ttc7HQ2zPcfjD136X8BAw?usp=sharing

reset() Function and Test

Next, we created a reset() function, which builds upon the homing function by additionally opening the gripper. This provides a clean and safe reset point for starting any future operation.



Screenshot: The defined reset() function in dobot.py

The corresponding test script (test_reset.py) performs as follows:

1. A full reset to the defined home position with the gripper opened

2. A movement to a defined target position.

3. Closes the gripper to simulate gripping an object

4. Calls reset() again to return and open the gripper

5. Closes the device

A Screenshot of the test script can be seen below

```python
from pydobot import Dobot
from serial.tools import list_ports
import time

port = list_ports.comports()[0].device
device = Dobot(port=port, verbose=True)

print("\n--- Homing to start ---")
device.reset()
time.sleep(2)

print("\n--- Moving to test position and closing gripper ---")
device.move_to(250.0, 50.0, 50.0, 0.0, wait=False)
device.grip(True)
time.sleep(2)

print("\n--- Resetting ---")
device.reset()
time.sleep(2)

device.close()
print("\n--- Reset test complete ---")
```

Screenshot: test_reset - Test script for the reset command.

Below is a link that shows the source code along with a video demonstration of the test script executing successfully:

https://drive.google.com/drive/folders/1p-d7WqaO72czH0e2QiFZe8MXxBTGIN7I?usp=sharing

pick() and place() Functions and Test

To simulate realistic robotic arms, we developed two new functions: pick() and place(). These enable the arm to approach an object, pick it up using the gripper, and place it elsewhere.

```python
def pick(self, x, y, z, r=0.0):
    """
    Move above pick point, descend, grip object, lift back up.
    """
    lift_z = z + 20
    self.move_to(x, y, lift_z, r, wait=False)   # Move above
    self.move_to(x, y, z, r, wait=False)        # Move to pick height
    self.wait(1000)
    self.grip(True)                            # Close gripper
    self.wait(1000)                             # Brief pause
    self.move_to(x, y, lift_z, r, wait=False)   # Lift object


def place(self, x, y, z, r=0.0):
    """
    Move above place point, descend, release object, lift back up.
    """
    lift_z = z + 20
    self.move_to(x, y, lift_z, r, wait=False)   # Move above
    self.move_to(x, y, z, r, wait=False)        # Move to place height
    self.wait(1000)
    self.grip(False)                           # Open gripper
    self.wait(1000)
    self.move_to(x, y, lift_z, r, wait=False)   # Lift away
```

Screenshot: Defined pick() and place() functions in dobot.py.

We validated these functions using the test_pickAndPlace.py script. This script works as follows:
1. Resets the robot to a known home position with the gripper open.

2. Defined pick and place coordinates.

3. Calls device.pick() to grab a red cube at the pickup location.

4. Calls device.place() to move and release the cube at the target location.

5. Resets the device again to complete the cycle

6. Closes the program.

Below is a screenshot of the test_pickAndPlace python program

```python
from pydobot import Dobot
from serial.tools import list_ports
import time

port = list_ports.comports()[0].device
device = Dobot(port=port, verbose=True)

print("\n--- Homing to start ---")
device.reset()
time.sleep(2)

# Define positions (adjust as needed)
pick_x, pick_y, pick_z = 200.0, 0.0, -20.0
place_x, place_y, place_z = 250.0, 50.0, -20.0

print("\n--- Picking up object ---")
device.pick(pick_x, pick_y, pick_z)
time.sleep(2)

print("\n--- Placing object ---")
device.place(place_x, place_y, place_z)
time.sleep(2)

print("\n--- Returning to home ---")
device.reset()

device.close()
print("\n--- Pick and place test complete ---")
```

Screenshot: test_pickAndPlace - Test script for the pick and place functions.

Below is a link that shows the source code along with a video demonstration of the test script executing successfully:

https://drive.google.com/drive/folders/1vl-P4R-2_7Vzjh-wz4_iPRqQTYvnics3?usp=sharing

Challenges and Next Steps

- Wait Behavior issue:

  Currently, all movement functions use wait=False due to issues with the DOBOT's command queue when wait=True is enabled. When wait=True, the arm freezes and fails to proceed. This is not ideal and leads to the use of manual sleep() and wait() calls to simulate safe command delays. Fixing this will improve safety and code readability

- Next Focus:

  Once the wait=True issue is solved, we will begin full integration of the voice recognition system with the DOBOT, enabling the robot to respond to voice commands like "pick", "reset", and "place."

6/01/2025 - 6/07/2025

6/01/2025

Adopting Final DOBOT SDK with Voice Control Implementation

Objective

Migrate to a more stable Python SDK for controlling the DOBOT Magician Lite, recreate essential movement functions, and develop a voice-controlled interface simulating a claw machine-style control system. Additionally, establish reliable device connection and prepare for upcoming conveyor belt automation.

SDK Transition and Setup

After discovering continued limitations with the previous libraries, we transitioned to a **new GitHub Repository** to use for the DOBOT:
https://github.com/sammydick22/pydobotplus/tree/master

This library proved to be significantly more stable - most notably, it resolved the issue with wait=True flag behavior. Command queuing now functions reliably, allowing smooth, safe execution and motion sequences without needing to manually delay execution with sleep() or redundant wait() calls.

Function Redefinition in New Repository

The core movement functions were rewritten using the new dobotplus SDK.

```python
def home(self):
    """Soft home - moves to a safe predefined center location."""
    self._set_queued_cmd_clear()
    self._set_queued_cmd_start_exec()
    self.move_to(x=200.0, y=0.0, z=70.0, r=0.0, wait=True)

def reset(self):
    """Returns to home and opens gripper."""
    self.home()
    self.grip(False)
```

Screenshot: Newly defined home() and reset() functions.

- home() clears and starts the command queue, then sends the robot to a defined center.
- reset() calls home() and opens the gripper.

```python
def pickOrPlace(self, x, y, z, r=0.0, do_pick=True):
    """
    Perform either a pick or a place operation:
    - Moves first in XY at a safe Z height.
    - Descends Z to target.
    - Grips or releases based on do_pick flag.
    - Lifts Z after action.
    """
    safe_z = 50.0  # Adjust if needed for your workspace

    # Get current position
    current = self.get_pose().position
    current_x, current_y, current_z, current_r = current.x, current.y, current.z, current.r

    # 1. If Z is low, lift to safe height before moving XY
    if current_z < safe_z:
        self.move_to(x=current_x, y=current_y, z=safe_z, r=current_r, wait=True)

    # 2. Move XY at safe Z
    self.move_to(x=x, y=y, z=safe_z, r=r, wait=True)

    # 3. Lower to target Z
    self.move_to(x=x, y=y, z=z, r=r, wait=True)

    # 4. Gripper action
    self.grip(do_pick)
    time.sleep(1)  # Short hold time

    # 5. Lift back to safe height
    self.move_to(x=x, y=y, z=safe_z, r=r, wait=True)

    print(f"[ACTION] {'Picked' if do_pick else 'Placed'} object at ({x}, {y}, {z})")
```

Screenshot: Newly defined pickOrPlace() function.

This generalized function performs either a pick or a place based on a do_pick flag, allowing safe vertical offsetting, target approach, gripping/release, and return.

```python
def auto_connect_dobot():
    print("[INFO] Scanning available serial ports...")
    ports = list_ports.comports()

    # First try auto-detected ports
    for p in ports:
        print(f"Checking port {p.device} (VID={p.vid}, PID={p.pid}, Name={p.name})")
        if "ttyACM" in p.device or "ttyUSB" in p.device:
            print(f"[INFO] Attempting to connect to DOBOT on port: {p.device}")
            try:
                return Dobot(port=p.device)
            except Exception as e:
                print(f"[WARN] Failed to connect on {p.device}: {e}")

    # Manual fallback if /dev/ttyACM0 exists
    if os.path.exists("/dev/ttyACM0"):
        print("[INFO] Using fallback: /dev/ttyACM0")
        try:
            return Dobot(port="/dev/ttyACM0")
        except Exception as e:
            raise Exception(f"Fallback failed on /dev/ttyACM0: {e}")

    raise Exception("DOBOT not found on any connected serial port.")
```

Screenshot: Defined auto_connect_dobot() function.

This function attempts to auto-detect the DOBOT's USB port or falls back to /dev/ttyACM0. This is now used in every script for robustness.

Voice-Controlled Claw Simulation

To test full system integration, we created a "claw machine" voice control program where each keyword triggers a specific robotic movement. The coded voice commands can be seen below:

```
if "up" in command:
    move_direction(dz=step_size)

elif "down" in command:
    move_direction(dz=-step_size)

elif "left" in command:
    move_direction(dy=-step_size)

elif "right" in command:
    move_direction(dy=step_size)

elif "forward" in command:
    move_direction(dx=step_size)

elif "backward" in command:
    move_direction(dx=-step_size)

elif "home" in command:
    print("[ACTION] Returning to home.")
    device.home()
    current_position.update({"x": 200.0, "y": 0.0, "z": 70.0, "r": 0.0})

elif "grip" in command:
    print("[ACTION] Gripper closing.")
    device.grip(True)

elif "release" in command:
    print("[ACTION] Gripper opening.")
    device.grip(False)

elif "quit" in command:
    print("[ACTION] Shutting down. Returning home.")
    quit_program = True
    device.home()
    break
```

Screenshot: Coded voice commands in test_voice_control.py.

This program requires a virtual environment (due to the microphone input), so to run the program you need to navigate to the project directory

$ cd Code/test/pydobotplus

And then run the following commands to activate the virtual environment and run the scripts:
$ source venv/bin/activate
$ python test_voice_control.py

Command-to-Action Mapping Table:

Voice Command	Robot Behavior
“up”	Moves arm up along Z-axis
“down”	Moves arm down along Z-axis
“left”	Moves arm left along Y-axis
“right”	Moves arm right along Y-axis
“forward”	Moves arm forward along X-axis
“backward”	Moves arm backwards along X-axis
“home”	Calls device.home() to return to safe center
“grip”	Closes the gripper
“release”	Opens the gripper
“quit”	Sends arm home and exits the script

Linked below is the source code for the script along with a demo video:
https://drive.google.com/drive/folders/1lRtJSxSa_nCYSD-T-pCW6OJc8iDaDWNr?usp=sharing

Side Note on Queued Command Handling

A key advantage of the pydobotplus library is its proper queuing mechanism. When wait=True is passed to a movement command, the SDK handles internal command timing cleanly without requiring external delays or manual timing. This is especially beneficial when chaining movements (like pick-lift-place sequences), ensuring safety, consistency, and simpler code maintenance.

Next Steps

- Begin integrating the DOBOT conveyor belt
  - Plan to write a color-sorting script, where colored blocks pass along the belt and the arm sorts them into groups
- Investigate intermittent hardware issues
  - Occasionally, the Raspberry Pi fails to recognize the DOBOT on boot
  - Temporary solution has been to reboot the Pi, but we plan to isolate this issue and implement a permanent fix.

6/05/2025

Transition to Whisper AI for Voice Recognition & Fuzzy Integration

Objective

Evaluate Whisper AI as an alternative to the Google Speech Recognition API, improve transcription accuracy using fuzzy matching, and begin troubleshooting conveyor belt integration with the DOBOT Magician Lite.

Whisper AI + Fuzzy Matching Test

We developed and tested a new Python script, whisper_voice_test.py, to evaluate the accuracy and performance of the Whisper AI speech recognition model. Unlike earlier programs, this script does not control the DOBOT directly - its only purpose is to test transcription quality in isolation.

```python
import sounddevice as sd
import numpy as np
import scipy.io.wavfile as wav
import os
import tempfile
from faster_whisper import WhisperModel
from fuzzywuzzy import process, fuzz
import signal
import sys

# --- Setup ---
MIC_INDEX = 0  # Change this to your TONOR mic index
COMMANDS = [
    "up", "down", "left", "right",
    "forward", "backward", "home",
    "grip", "ungrip", "release",
    "quit", "exit", "stop"
]
RECORD_DURATION = 2  # seconds
SAMPLE_RATE = 16000

model = WhisperModel("base.en", compute_type="auto")  # Can change to 'tiny.en' for speed

# --- Signal handler for Ctrl+C ---
def signal_handler(sig, frame):
    print("\n[INFO] Ctrl+C detected. Exiting.")
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
```

Screenshot: Import and Setup for whisper_voice_test.py.

```python
# --- Function to record and transcribe ---
def record_audio(duration=RECORD_DURATION, samplerate=SAMPLE_RATE):
    print(">> Speak now...")
    recording = sd.rec(int(duration * samplerate), samplerate=samplerate, channels=1, dtype='int16', device=MIC_INDEX)
    sd.wait()
    return recording.squeeze()

def save_temp_wav(audio, samplerate=SAMPLE_RATE):
    temp_path = tempfile.NamedTemporaryFile(delete=False, suffix=".wav")
    wav.write(temp_path.name, samplerate, audio)
    return temp_path.name

def transcribe_with_whisper(wav_path):
    segments, _ = model.transcribe(wav_path, language="en")
    for segment in segments:
        return segment.text.strip().lower()
    return ""

def fuzzy_match(command):
    match, score = process.extractOne(command, COMMANDS, scorer=fuzz.ratio)
    return (match, score) if score > 70 else (None, score)
```

Screenshot: Functions for recording & transcribing in whisper_voice_test.py.

```python
# --- Main loop ---
if __name__ == "__main__":
    print("[VOICE] Listening for:", ", ".join(COMMANDS))

    while True:
        try:
            audio = record_audio()
            wav_path = save_temp_wav(audio)
            transcription = transcribe_with_whisper(wav_path)
            os.remove(wav_path)

            if not transcription:
                print("[WARN] No speech detected.")
                continue

            print(f"[VOICE] You said: '{transcription}'")
            match, score = fuzzy_match(transcription)

            if match:
                print(f"[✓] Recognized as '{match}' (confidence: {score}%)")
                if match in ["quit", "exit", "stop"]:
                    print("[INFO] Exit command received. Shutting down.")
                    break
            else:
                print(f"[X] No valid match found (confidence: {score}%)")

        except Exception as e:
            print(f"[ERROR] {e}")
```

Screenshot: Main Loop in whisper_voice_test.py.

Test results showed significantly better performance than the Google API, especially in noisy conditions. Due to this, **Whisper AI and fuzzy matching will now be the default speech recognition method** going forward.

We also explored adding a prefix keyword to voice commands (e.g., "Dobot right") to verify device targeting. However, "Dobot" was often mis-transcribed. The term "Magic" was chosen instead, as it transcribes more reliably. For example: **“**Magic right**”** correctly triggers right movement with much higher consistency.

Conveyor Belt Troubleshooting

To begin conveyor belt integration testing, we created a Python script called test_conveyor.py using the pydobotplus library.

```python
import time
import sys
from external.pydobotplus import auto_connect_dobot

# --- Connect DOBOT ---
device = auto_connect_dobot()

try:
    # --- Optional: Reset/initialize state ---
    device.speed(velocity=100, acceleration=100)
    device.home()
    print("[INFO] Homed and ready.")

    # --- Run conveyor forward (STP2) ---
    print("\n[TEST] Running conveyor forward (STP2)...")
    device.conveyor_belt(speed=0.5, direction=1, interface=0)
    time.sleep(5)

    # --- Stop ---
    print("[TEST] Stopping conveyor...")
    device.conveyor_belt(speed=0.0, interface=0)
    time.sleep(2)

    # --- Run conveyor backward (STP2) ---
    print("[TEST] Running conveyor backward (STP2)...")
    device.conveyor_belt(speed=0.5, direction=-1, interface=0)
    time.sleep(5)

    # --- Stop ---
    print("[TEST] Stopping conveyor...")
    device.conveyor_belt(speed=0.0, interface=0)

finally:
    device.close()
    print("[INFO] Conveyor test complete. Device disconnected.")
```

Screenshot: The test_conveyor.py script.

Despite testing every interface configuration, the script did not activate the conveyor belt.

We then attempted control using DobotLab with a block-based program following the official curriculum documentation:
https://www.robotlab.com/hubfs/V1.1%20Dobot%20Lite%20Curriculum%202024.pdf?srsltid=AfmBOoqAWFIVro3wdGbbMuQhWL4xRmK7AdDct4qEHE7EjaHkGUJ-Zw-Q

Screenshot: Block Code for DobotLab.

However, this also failed to trigger any movement.

At this point, based on consistent failure across both software environments, we suspect a hardware-related issue with the conveyor motor or connections. Further physical inspection and component swapping will be done next session to isolate the problem.

6/08/2025 - 6/14/2025

6/09/2025

We have successfully got the conveyor to run. Well, sort of. The issues we were having with DobotLab seemed to be resolved once disconnecting the color sensor. Once disconnected the conveyor script above worked, along with the Infrared Sensor! This was a great step forward but we then wanted to see if the Python Scripts would now work.

With that being said, the Python scripts are still failing to work using pydobotplus. At this point we are not really sure what is causing this issue still but we have eliminated the possibility of it

being a hardware issue. Nitesh is currently looking into the issue and we hope to have this working very soon.

6/12/2025

Conveyor Belt Control on Windows

The objective with the conveyor belt this week was to get it running via Python. We successfully ran the DOBOT conveyor using Python on a Windows machine. This was achieved using a DLL-based control package shared by Dr. Zhou: https://www.dropbox.com/scl/fi/2s5348ll3k7fha8qtq7rj/demo-magician-python-64-master.zip?rlkey=drvm32v1mv3ad81ydg9ddhzjv&dl=0

We created a test script titled conveyor_windows.py. This script loads the DLL and establishes connection, then sends forward/reverse commands to the conveyor using SetEMotor(), successfully controlling the conveyor with high-speed values. This script can be seen below.

conveyor_windows.py > ...

```python
import sys
import os
import time

# Add dobot_magician to the import path
sys.path.append(os.path.join(os.path.dirname(__file__), "dobot_magician"))
import DobotDllType as dType

# Load DLL
api = dType.load()

# Connect to Dobot
state = dType.ConnectDobot(api, "", 115200)[0]
print("Connection status:", state)

if state == dType.DobotConnect.DobotConnect_NoError:
    dType.SetQueuedCmdClear(api)
    dType.SetQueuedCmdStartExec(api)

    print("Running conveyor forward...")
    dType.SetEMotor(api, 0, 1, 7000)  # index=1, isEnabled=1, speed=1000
    time.sleep(5)

    print("Stopping conveyor...")
    dType.SetEMotor(api, 0, 0, 0)
    time.sleep(1)

    print("Running conveyor in reverse...")
    dType.SetEMotor(api, 0, 1, -7000)  # negative speed = reverse
    time.sleep(5)

    print("Stopping conveyor...")
    dType.SetEMotor(api, 0, 0, 0)

    dType.SetQueuedCmdStopExec(api)
    dType.DisconnectDobot(api)
```

Screenshot: conveyor_windows.py script to run the conveyor.

A link to the source code along with a short **demo video** showing the conveyor in action can be seen below:
https://drive.google.com/drive/folders/1DstKMFN9sH5pLZzHzPsHwvKEszxz3F8q?usp=sharing

We now plan to investigate how to adapt this for Linux (Raspberry Pi). We suspect that with access to the DLL source code, we might be able to create or wrap a compatible .so shared object for linux use, but we will look more into this in the near future.

Multi-Robot Voice Command Framework

Another objective for this week was to begin integrating voice-based command execution for both the DOBOT Magician Lite and UR5e arms using a shared script. We began constructing a unified voice control system that responds to robot-specific commands. Each robot is addressed by a keyword prefix:

- DOBOT Magician Lite is referred to as “magic”
- UR5e is referred to as “robo”

We created the file robot_collaboration.py to capture microphone input and transcribe using Whisper, then match commands using fuzzy logic, then route the commands based on the robot name prefix. The current command phrases will later be altered to commands that will lead to assembly, disassembly, etc. But this will be done later on when we get YOLO implemented.

```
COMMANDS = [
    # DOBOT
    "magic up", "magic down", "magic left", "magic right", "magic forward", "magic backward",
    "magic home", "magic grip", "magic ungrip", "magic release",
    "magic pick from tray", "magic place on table",

    # UR5e
    "robo move up", "robo move down", "robo home", "robo stop",

    # Exit
    "quit", "exit", "stop"
]
```

Screenshot: Current Command Phrases

While the UR5e arm responds consistently and correctly to commands like “robo move up” and “robo stop”, the DOBOT’s behavior is inconsistent. At times, it runs the correct command, while other times it silently fails even when the transcription and function calls are accurate.
A full **demo video** and the current version of the script can be seen using the link below (A new fully working demo can be seen later on in the memo on 6/16):
https://drive.google.com/drive/folders/1v8tHa6zjQEZEc9nBZ8CAkTvHASi7363M?usp=sharing

Another known issue deals with unstable ethernet connection to the UR5e. The Ethernet connection between the Raspberry Pi and the UR5e robot drops randomly. After a few minutes, the Pi is unable to ping the UR5e. Temporarily, we resolve this by unplugging and replugging the Ethernet cable. The root cause is unknown and will require further troubleshooting — possibly involving power management or IP leasing.

6/15/2025 - 6/27/2025

6/16/2025

Multi-Robot Voice Command Framework cont…
After some debugging, we successfully resolved the issue with the robot_collaboration.py script. The earlier inconsistencies, particularly with the DOBOT not executing commands, were traced back to hardware-level disconnections during runtime. These caused the device to fail silently, even when transcription and command logic were verified to be correct.
The current version of the script now functions reliably, and the demo confirms:

- Voice input triggers action on both the DOBOT magician lite and the UR5e.
- The system supports multi-robot input in a single script session.
- The DOBOT now responds more consistently due to improved connection stability (see below)

While the command vocabulary is still basic (e.g. “robo move up” or “magic grip”), this architecture lays the foundation for future high-level functions like “assemble gear” or “start disassembly.”
The source code along with a **demo video** can be seen via the link below:
https://drive.google.com/drive/folders/1-F1-BPFwMEYGkpCfDqIQvRyGH81WDaVW?usp=sharing

Improving Device Connection Issues

For robust operation, we needed to eliminate two major hardware communication issues: (1) the DOBOT randomly disappearing from USB, and (2) the UR5e losing Ethernet connection after a few minutes. Both problems have now been resolved.

Dobot USB Dropout Fix
The issue here was that the DOBOT Magician Lite would vanish from lsusb after running for some time, requiring a full Raspberry Pi reboot to re-establish communication. To diagnose this problem, I ran the command: $ dmesg | grep usb, to show the EMI-related USB resets. Although autosuspend was disabled globally, individual USB devices still defaulted to auto.
To solve this issue I created a udev rule to force persistent USB power for the DOBOT:

- Running the following command:
  $ *sudo nano /etc/udev/rules.d/99-dobot-usb.rules*
- And adding the following to the file:
  ACTION=="add", ATTR{idVendor}=="f055", ATTR{idProduct}=="9800", TEST=="power/control", ATTR{power/control}="on"
- Then reloading the rules by calling the following commands:
  $ *sudo udevadm control --reload-rules*, and $ *sudo udevadm trigger*

The result was that the DOBOT device now stays connection consistently during long development sessions, with not reboot necessary

UR5e Ethernet Dropout Fix

The issue with the UR5e is that the arm would drop Ethernet communication after ~2 minutes. Pings would begin to fail, and the RTDE interface reported errors. To diagnose this issue I ran a command (journalctl) to show NetworkManager repeatedly reinitializing the interface. Ethtool showed the pi defaulting to 1000Mb/s with auto-negotiation, which proved unstable.
The solution here was to manually enforce a stable ethernet speed using the following command:

```
sudo ethtool -s eth0 speed 100 duplex full autoneg off
```

Screenshot: Command to force stable ethernet speeds.

I then made the setting persistent with a systemd service:

```
sudo nano /etc/systemd/system/force-eth0-speed.service
```

ini Copy

```
[Unit]
Description=Force eth0 to 100Mb/s Full Duplex
After=network.target

[Service]
Type=oneshot
ExecStart=/sbin/ethtool -s eth0 speed 100 duplex full autoneg off
RemainAfterExit=true

[Install]
WantedBy=multi-user.target
```

Screenshot: Command and added content to make settings persistent.

Then enabling the service and updating the IP settings using nmcli:

```
sudo systemctl daemon-reexec
sudo systemctl enable force-eth0-speed.service
```

```
sudo nmcli connection modify "UR5e and Scanner" ipv4.method manual ipv4.addresses 192.168.1.20/24 ipv4.gateway "" ipv4.dns ""
sudo nmcli connection down "UR5e and Scanner"
sudo nmcli connection up "UR5e and Scanner"
```

Screenshots: Commands and added content to enable the service.

The results showed that the Ethernet connection is now stable. Ping tests show 0% packet loss over extended sessions, and the RTDE control interface runs without interruption.

Upcoming Tasks:

- Configuring the conveyor program to work with LINUX (in progress)
- Create a script utilizing the DOBOT, UR5e, and conveyor that performs a manufacturing task based on voice command input. Where an event from one of the robots triggers an event in the other.(Should get completed this week.)
- Once the YOLO training is completed, we will incorporate that into the script so that the robots can perform more complex tasks.

For Meeting on 6/25:

I introduced the robots and some of the workings behind them to Carter, I also sent a detailed email that lists some of the links and steps that will help him going further.

Before leaving on vacation, we have been struggling on getting C++ DOBOT files to work on linux. Me and Owen, worked on it for a couple of days and haven't had much luck. The shared library I compiled on the Pi connects to the robot and loads successfully, and while control commands return success codes, they have no effect on the hardware. We are kind of at a loss, but still working on getting this to work. - 6/19

6/26/2025
Integrated Disassembly Pipeline - Vision and Next Steps

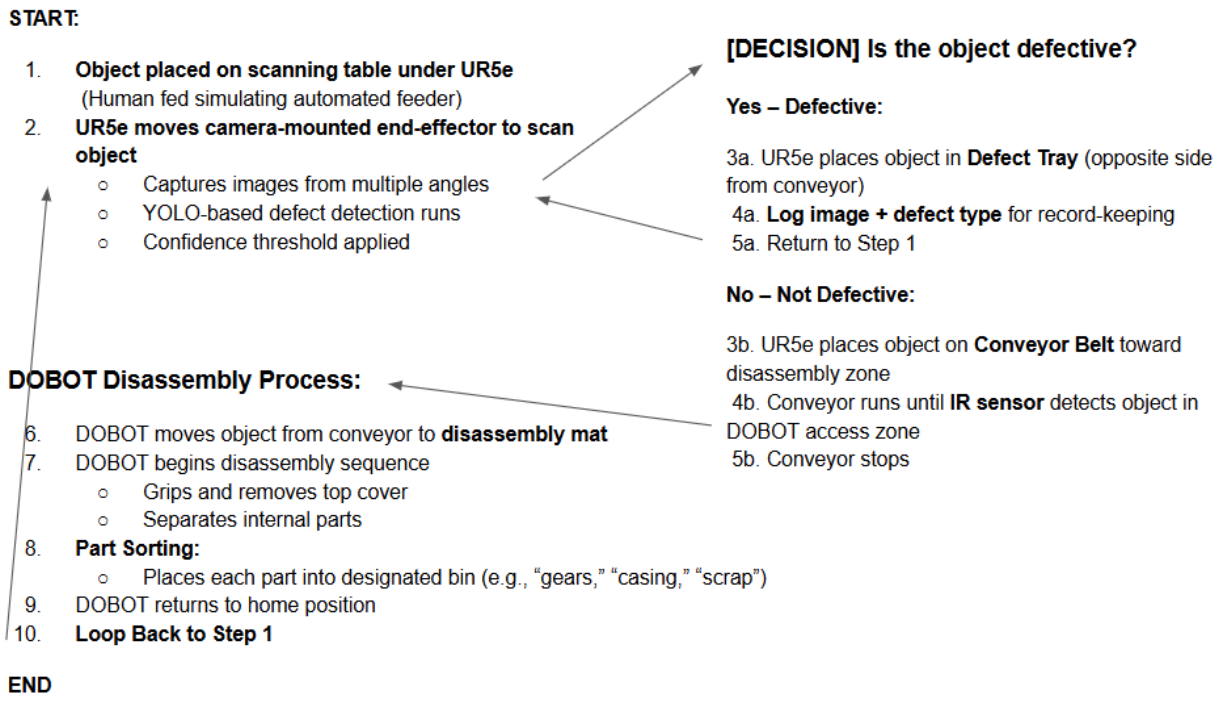
Overview:

With the UR5e and DOBOT Magician Lite physically integrated, and voice control systems stabilized, we now shift toward implementing a collaborative robotic program for gearbox disassembly. This chapter outlines the upcoming integration plan, supported by a visual flowchart and staged milestones.

The system will involve:

- **UR5e** for scanning, classification, and routing.
- **YOLO** (currently being trained) for defect detection.
- **Conveyor + IR sensor** for positioning handoff.
- **DOBOT** for disassembly and part sorting.

Flow Chart:



Screenshot: Gearbox Disassembly Flowchart.

Checkpoint Milestone Plan

Each checkpoint represents a functional milestone. Completing each stage ensures the program is progressing toward full end-to-end automation:

Checkpoint 1 - UR5e Scan + YOLO Detection Integration

- Load YOLO-trained model and verify it runs on UR5e-mounted camera feed
- Apply confidence threshold logic (e.g. >85% = pass, <85% = defect)
- Route outputs to defect classification or queue for next stage.
- Begin snapshot logging (image + defect class + confidence %)

Checkpoint 2 - UR5e Pick-and-Place to Conveyor/Defect Tray

- Program UR5e to:
    - Pick from scanning table
    - Place defective items in defect tray
    - Place non-defective items onto conveyor belt for disassembly
- Test repeatability and placement accuracy

Checkpoint 3 - Conveyor and IR sensor Coordination

- Enable conveyor motor control
- Calibrate IR sensor to detect when object is in DOBOT zone
- Write logic to pause conveyor when object is in position
- Test with a couple items in sequence

Checkpoint 4 - DOBOT Pickup and Disassembly Mat Placement

- Train DOBOT to:
    - Grab item from conveyor
    - Place it on a fixed location disassembly mat (Was thinking a rubber mat)
- Validate consistency and grip quality

Checkpoint 5 - DOBOT Disassembly + Sorting Routine

- Create a DOBOT movement sequence to:
    - Remove top housing
    - Extract interior parts in steps
    - Place parts in labeled bins (e.g., “gears,” “scrap”)
- Use jigs or physical guides if needed

Checkpoint 6 - End-to-End System Trial

- Run full pipeline from human placement to DOBOT sorting
- Use 2-3 test gearboxes, with some defective ones maybe
- Log classification data, timing metrics, and DOBOT performance

Checkpoint 7 - Logging + Monitoring Infrastructure

- Save image snapshots from YOLO
- Maintain defect records (type, location, confidence)
- Could also log DOBOT disassembly completion or part count per object

Final Considerations

- Voice Overrides: Future versions can add voice-based intervention if disassembly fails

- Error Handling: Add retries or fallbacks if IR sensor doesn't detect movement within expected time
- Component Reuse: Sorted parts may be reused or tracked in inventory depending on future expansion

6/27/2025

Dobot Logging System Development and Integration

Overview:
This chapter documents the development of a robust logging system for the DOBOT Magician Lite using the pydobotplus library after an initial evaluation of the LeRobot framework. The goal was to enable repeatable, human-readable pose logging and test its integration into future control scripts. We developed a reusable Python module and verified functionality through multiple logging and motion demos.

Abandoning LeRobot for pydobotplus:
I initially explored the LeRobot framework as a potential unified system for managing robotic arms. However, after cloning and inspecting the GitHub Repository (https://github.com/huggingface/lerobot), we determined that it did not contain support for either the DOBOT Magician Lite or the UR5e. Grep searches for "Dobot" and "UR5" returned no usable device interface modules. Consequently, we opted to build our logging system around the existing and functional pydobotplus SDK, which had already proven reliable in prior voice-controlled motion work.

Initial Demo: dobot_motion_logging_demo.py
Our first goal was to verify that we could consistently log the DOBOT's position over time into a CSV file. We created a demo script, *dobot_motion_logging_demo.py*, which established a serial connection to the DOBOT, performed a simple motion, and saved time-stamped joint and position data into a CSV file.
The purpose of this script was:

- Ensure get_pose() returns reliable data at regular intervals
- Demonstrate the ability to format and save logs cleanly for presentation or analysis

The script was verified via terminal output and log inspection. A sample run showed the arm performing a simple movement while logging to *dobot_motion_log_demo.csv.* A screenshot can be seen below of the terminal output.

```
[*] Connecting to DOBOT...
[✓] Connected.
[*] Logging motion to: dobot_motion_log_demo.csv
[*] Moving forward 50mm in Y...
[*] Moving back to home position...
[✓] Motion and logging complete.
(.venv) stem@cvpi:~/Code/test/logging $ cat dobot_motion_log_demo.csv
timestamp,x,y,z,r,j1,j2,j3,j4
2025-06-27T16:10:04.214202,250.0,100.0,50.0,0.0,21.8,22.7,36.96,-21.8
2025-06-27T16:10:04.756214,250.0,150.0,50.0,0.0,30.96,29.77,33.12,-30.96
2025-06-27T16:10:05.262282,250.0,150.0,50.0,0.0,30.96,29.77,33.12,-30.96
2025-06-27T16:10:05.768285,250.0,150.0,50.0,0.0,30.96,29.77,33.12,-30.96
2025-06-27T16:10:06.808330,250.0,100.0,50.0,0.0,21.8,22.7,36.96,-21.8
2025-06-27T16:10:07.314388,250.0,100.0,50.0,0.0,21.8,22.7,36.96,-21.8
2025-06-27T16:10:07.820401,250.0,100.0,50.0,0.0,21.8,22.7,36.96,-21.8
(.venv) stem@cvpi:~/Code/test/logging $
```

Screenshot: Terminal Output Of dobot_motion_logging_demo.py

The source code along with a short **demo video** are linked here:

https://drive.google.com/drive/folders/1da_nft2rfyiHc_5tOlL920jjucL7xpcp?usp=sharing

Reusable Module: dobot_logger.py

After confirming the logging pipeline worked reliably, we developed a reusable Python module called *dobot_logger.py.* This class-based system is built around a separate thread that logs pose data at a configurable interval and writes to a cleanly formatted CSV file.

Key Features:

- Automatic timestamped filename generation
- Clean header formatting (timestamp, x, y, z, r, j1, j2, j3, j4)
- Graceful start/stop methods
- Designed to be easily importable into any DOBOT script

The logger handles threading internally, keeping the main script responsive even while writing log entries. The logger proved robust across multiple tests.

Integration Test: logging_in_action.py

To demonstrate how easily *dobot_logger.py* can be integrated into new projects, we created a motion script titled *logging_in_action.py.* This script:

- Connects to the DOBOT using pydobotplus
- Instantiates the logger and begins logging to the /logging directory
- Sends the DOBOT through a multi-step sequence
- Stops logging after motion completes

This test proved:

- The logger can operate in parallel with movement commands
- Log files are stored in a readable format
- Integration requires minimal code changes

The terminal output of this script can be seen below:

```
(.venv) stem@cvpi:~/Code/test/logging $ python logging_in_action.py
[*] Connecting to DOBOT...
[✓] Connected.
[*] Started logging to: dobot_log_2025-06-27_16-52-45.csv
[*] Logging is now active.
[*] Performing motions while logging...
[✓] Logging stopped.
[✓] Logging stopped.
[✓] Motion and logging complete.
(.venv) stem@cvpi:~/Code/test/logging $ cat dobot_log_2025-06-27_16-52-45.csv
timestamp,x,y,z,r,j1,j2,j3,j4
2025-06-27T16:52:45.102080,259.24,81.90,49.80,-23.82,17.53,23.55,36.60,-41.36
2025-06-27T16:52:46.108142,259.24,131.90,49.80,-23.82,26.97,29.58,33.29,-50.79
2025-06-27T16:52:47.114058,305.87,132.28,50.00,-23.82,23.29,44.35,22.96,-47.11
2025-06-27T16:52:48.120112,309.24,131.90,49.80,-23.82,23.10,45.11,22.43,-46.93
2025-06-27T16:52:49.128151,309.24,131.90,79.80,-23.82,23.10,41.24,13.05,-46.93
2025-06-27T16:52:50.134225,293.15,111.54,67.76,-23.82,20.53,33.07,23.93,-44.35
2025-06-27T16:52:51.140288,259.24,81.90,49.80,-23.82,17.53,23.55,36.60,-41.36
```

Screenshot: Terminal output of *logging_in_action.py*

Both the *dobot_logger.py* and *logging_in_action.py* source scripts along with a short **demo video** of this logging integration here: https://drive.google.com/drive/folders/1Sm4cPU0gxWvfHcQ8aGtWxyWFMZZn-SZY?usp=sharing

Conclusion:

With the creation and testing of both *dobot_motion_logging_demo.py* and *dobot_logger.py,* we now have a stable, reusable logging system for all future DOBOT projects. This enables us to benchmark, debug, or analyze robot pose data efficiently and reliably, whether during motion testing, pick-and-place trials, or integrated disassembly pipelines. The module will be included as a core utility in all future scripts that involve DOBOT movement tracking or diagnostics.

6/29/2025 - 7/12/2025

7/01/2025

Gocator Scanning and Multi-Robot Transport Coordination

This chapter focuses on the development and successful implementation of scanning_demo.py, a script that coordinates the DOBOT and UR5e arms to perform object transfer, scanning, and placement using a conveyor belt and a Gocator 3D sensor. The goal was to have the DOBOT arm pick up an object using a suction tool, place it on a conveyor, then have the UR5e arm scan and remove the object from the other end. Scan data and image files would be saved automatically.

System Goals and Workflow
The complete workflow performs the following sequence:

- DOBOT uses suction to pick up a flat object and place it onto a conveyor.
- The conveyor runs for a calibrated time to bring the object to a precise scanning position.
- The UR5e arm moves into a scanning position, where the Gocator captures 3D data and image files.
- The UR5e then picks up the scanned object and places it onto a designated space on the table.
- All scanner outputs - including .ply mesh files and .png images - are saved in local directories, along with printed measurement readouts.

Script Development Path
To arrive at the final system, multiple supporting scripts were created and used:

- Dobot_find_conveyor_end.py: This script allowed for precise experimentation with the DOBOT suction gripper. It helped identify reliable pick and place coordinates on the conveyor and verified suction actuation using dobot.grip(). This early calibration was essential before any full pipeline testing.
- Ur5e_coordinate_logger.py: Provided by Nitesh, this script allowed me to log accurate UR5e Cartesian coordinates by placing the robot into local mode (for physical manipulation) and then reading coordinates back in remote mode. It was used to record three key positions: the scanning position, pickup location, and final placement spot.
- Conveyor_alignment_test.py: This script stitched together DOBOT pick/place motion and conveyor control to test timing and alignment. The Gocator live stream was used to visually verify whether the object landed in an optimal scanning position. Conveyor run time was iteratively tuned to ~3.6 seconds based on visual inspection of final alignment.

All of these scripts were important stepping stones in developing the final working system. Their source code can be seen via the link below:
https://drive.google.com/drive/folders/13fEDIa-5VxjgOQ-C9BRMamfDwEaqxJL_?usp=sharing

Final Working Script: scanning_demo.py
The final script combines all previous components into a single pipeline. It uses pydobotplus to control DOBOT motion and suction, DoBotArm to control the conveyor, and the UR5e RTDE

SDK to move between calibrated scan and pick/place locations. Gocator SDK bindings written in CTypes allow for automatic capture and saving of scan data.
Scan results include:

- .ply surface mesh files (3D structure)
- .png images (Surface Z projection)
- Raw and processed measurement data (angles, positions, height)

All outputs are stored in organized folders below the working directory.Final source code and a working **demo video** can be seen via the link below:
https://drive.google.com/drive/folders/1pK9qlNWp4yYKfIUHRaL1ro3xlfWBX500?usp=sharing

The terminal output after a successful run can also be seen below:

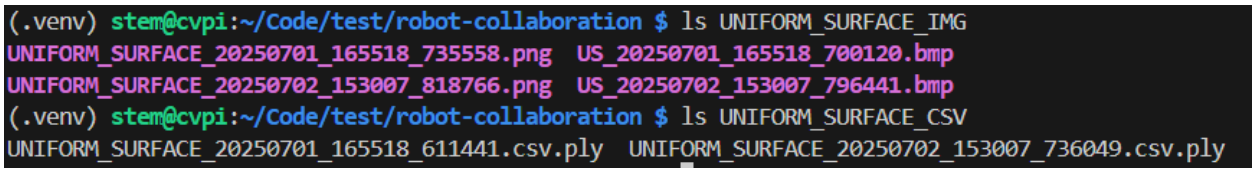
```
(.venv) stem@cvpi:~/Code/test/robot-collaboration $ python scanning_demo.py
[INFO] SDK libraries loaded successfully.
[*] Connecting to UR5e...
[*] Moving UR5e to home position...
[*] Connecting to DOBOT...
[*] Picking up object...
[*] Above pickup → x:210, y:0, z:37, r:0
[*] Lowering to pickup → x:210, y:0, z:-33, r:0
[*] Lifting → x:210, y:0, z:37, r:0
[*] Placing on conveyor...
[*] Above dropoff → x:338, y:147, z:100, r:0
[*] Lowering to dropoff → x:338, y:147, z:30, r:0
[*] Retreating → x:338, y:147, z:100, r:0
[*] Returning DOBOT home...
[*] Running conveyor...
Connect status: DobotConnect_NoError
[*] Moving to scan position...
[*] Scanning object with Gocator...
Surface Message
Surface data width: 737
Surface data length: 395
Total num points: 291115
[*] Waiting for scan data...
PLY file saved at: UNIFORM_SURFACE_CSV/UNIFORM_SURFACE_20250702_153007_736049.csv.ply

Image saved at: UNIFORM_SURFACE_IMG/UNIFORM_SURFACE_20250702_153007_818766.png

Measurement Message batch count: 1
Measurement ID:  1
Measurement Value: 7.601 mm
Measurment Decision: 1

[*] Picking object with UR5e...
[*] Lifting object slightly...
[*] Placing on desk...
[*] Moving UR5e to home position...
[√] Scanning and placement complete.
[*] Closing DOBOT connection.
[*] Stopping UR5e script.
```

Screenshot: Terminal Output Of scanning_demo.py.



Screenshot: Saved files from scan in scanning_demo.py

7/03/2025

Implementing Voice Control

WIth the scanning_demo.py script working now, I wanted to create a script that can perform the same functions but be controlled using voice. I wanted a program that had a push-to-talk feature where I could control the DOBOT and ur5e's functions. Before working on the program I thought out some good command ideas. These ideas include:

- "Run full cycle" : Runs the same functions as scanning_demo.py, where the DOBOT picks up an object off the table, places it on the conveyor's end, where it then runs till reaching the ur5e scan position, where the data from the scan is then saved and the ur5e can then pick and place the object on the desk.
- "End cycle": Quits the program mentioned above
- "Magic Home": Calls the DOBOT arm to a home position
- "Robo Home": Calls the ur5e to a home position
- "Test Magic Suction": Runs the suction on the DOBOT for 2 seconds then quits
- "Test Robo Suction": Runs the suction on the ur5e for 2 seconds then quits
- "Log Robo Pose": Gives the ur5e's current coordinates in the terminal
- "Test Camera": Does a scan where the data is then saved
- "Scan Pose": Brings the ur5e to the scanning position (good for setting alignment)
- "Reset System": Sends both the ur5e and DOBOT arms to a home position
- "Shutdown": Shutdowns the overall program safely

This program, titled voice_scanning.py ended up working quite well. Currently I have the push to talk key set as the spacebar and when you want to give a command you just hold down the spacebar while you're giving the command, and let go when you're done, the command will then be executed!
In the link below you can find the source code, the launch file, and a short **demo video** showing some of the scripts functions:
https://drive.google.com/drive/folders/1Gda0NRdiOgLzybTlrdy6v6iLR6SAbG3J?usp=sharing

Some Challenges Seen:

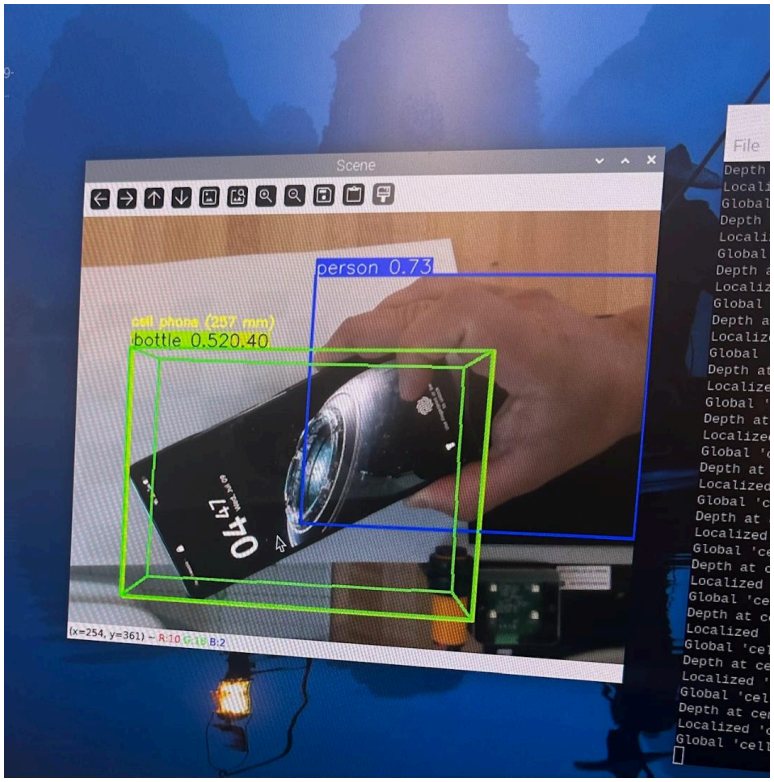
- Segmentation Fault - sometimes the program will run into a segmentation error, I believed this was due to the scanner closing improperly and I have made the proper changes and it occurs less, but every now and then it will still occur. This is definitely a problem we would want fully resolved before the final script is in place.
- The ur5e's suction - The ur5e's suction only activates for a small amount of time when start_suction() is called from epick_gripper.py. This has been an issue that Nitesh has told us about for a while, this also definitely needs resolved, and some of the epick_gripper.py script may need to be changed

7/09/2025

Brief Object Detection Update

I have been specifically looking into object detection. We have made a little progress on this part. What has been done is documented here:

- Installed pyrealsense2 - this took an ample amount of time for some reason, the downloads took about 15-20 minutes each, I also had to download python 3.9 for a virtual environment to be able to use this.
- Calibrated the camera - We ran both the collect_data.py script and hand_eye_callibration.py scripts and received the necessary files generated from them
- Worked on creating a 3-D bounding box - This is currently still in the works, we have had some okay results but nothing that great, an example can be seen below:



Screenshot: 3D bounding box example.

I plan on working on this more throughout this and next week. We are currently using a random YOLO model (hence why in the image the model recognizes a phone as a bottle.), but we plan to incorporate ours once its more polished.

7/13/2025 - 7/26/2025

7/14/2025

Oriented 3D Bounding Boxes

As part of improving object understanding for robotic manipulation, we began working on a robust method to generate 3D Oriented Bounding Boxes (OBBs) using the RealSense camera and YOLO detection, with the goal of outputting accurate 6D poses (position + orientation) aligned to the UR5e robot's base frame. Several approaches were explored and tested both visually and mathematically to determine which one could be used in a real-time robotic system.

Objective

Real-time detection and visualization of true 3D Oriented Bounding Boxes around objects (e.g. gears or a mouse (testing)) using YOLO + RealSense depth -> 3D projection -> coordinate transformation into the UR5e base frame.

Method 1: Open3D OBB via get_oriented_bounding_box()
What we used:

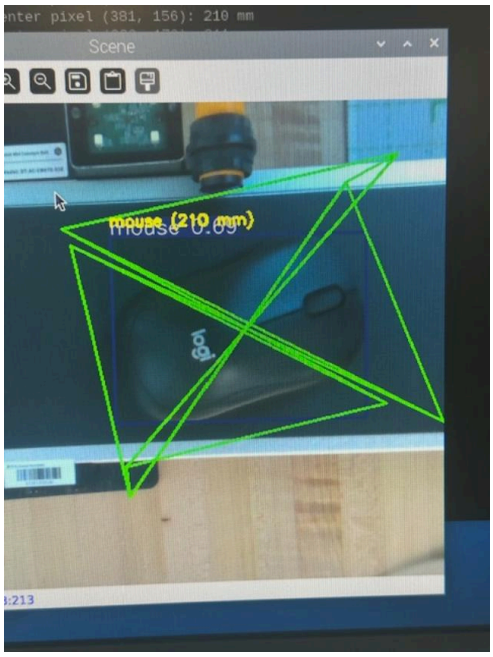
- Constructed a 3D PointCloud using Open3D
- Called .get_oriented_bounding_box() to estimate the box

Why it seemed useful:

- Robust geometry engine with tested 3D math.
- Automatically computes tight-fitting boxes.
- Built-in 3D visualization for quick debugging

Downsides:

- Requires OpenGL and GUI backend, which breaks in headless environments like Raspberry Pi.
- Incompatible with Pi's Python 3.9 OpenCV stack.
- Could not integrate bounding box drawing with OpenCV windows.



Screenshot: 3D Bounding Box Made With Open3D

Method 2: PyOBB (pyobb.OBB)
What we used:

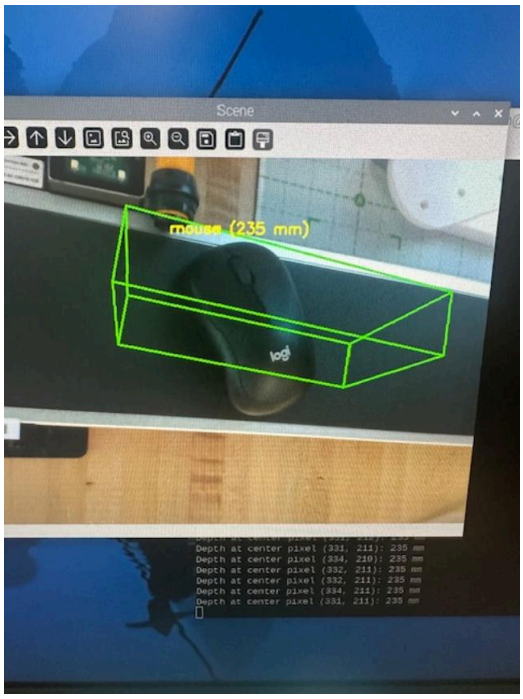
- Used OBB.build_from_points(points) to compute centroid, orientation, and extent.
- Projected the corners to image space manually for drawing.

Why it seemed useful:

- Lightweight and geometry-specific.
- Offers orientation matrix and dimensions cleanly.

Downsides:

- Did not return corners in consistent order for drawing.
- Alignment errors with noisy or sparse depth data
- Boxes were mathematically valid but visually incorrect in most frames.



Screenshot: 3D Bounding Box Made with PyOBB

Method 3: Custom NumPy PCA

What we used:

- Performed PCA manually:
  - Centered point cloud
  - Computed covariance matrix
  - Extracted eigenvectors (orientation) and projections
- Reconstructed box using min/max bounds along principal axes.

Why it worked best:

- Compatible with pyrealsense2, OpenCV, and custom math.
- Allowed full control and debugging over orientation and corner reconstruction.
- 3D point cloud came from YOLO -> RealSense depth -> pixel-to-point backprojection.
- Output could be directly converted to a 6D pose and transformed into robot base frame using T_cam2gripper.

Downsides:

- Still susceptible to inaccuracies with flat, reflective, or small objects
- No outlier rejection or point cloud smoothing.

Final Integrated Approach: PCA + 6D Pose Projection

This is the approach we settled on for integration with our real robot system.

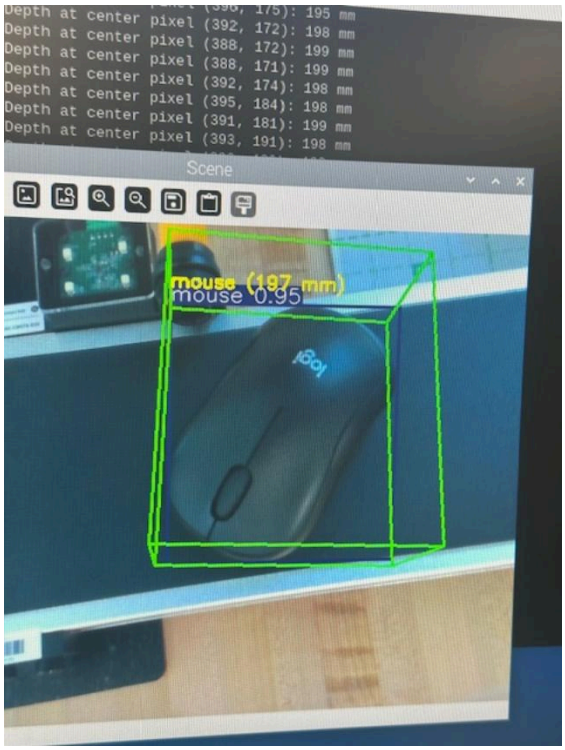
Why this worked best:

- Fully compatible with Raspberry Pi 5 (headless, low power).
- Real-time capable
- Visual overlays in OpenCV
- Pose computed as:
    1. 3D center + rotation matrix from PCA
    2. Transformed with T_cam2gripper
    3. Result expressed in UR5e base frame

Script capabilities:

- Object detection using YOLO
- 2D bounding boxes expanded into 3D using RealSense depth
- Oriented bounding box drawn on screen in real-time
- Top-center converted into a 6D pose
- Text overlay shows full (x, y, z, roll, pitch, yaw)
- Easily expandable to guide robotic actions like grasping or inspection

The source script can be seen using the link below:
https://drive.google.com/drive/folders/100qRnRp2izi90oMmep9wS3P7J2IFphXw?usp=sharing



Screenshot: 3D Bounding Box Made with PCA

Integration with UR5e Robot

Although the object detection and 6D pose estimation is now stable, full integration with the robot is still in progress, the plan is to:

- Use the computed 6D pose to define grasping or scanning locations
- Synchronize the bounding box result with the robot's TCP pose from rtde_receive

- Expand logic to allow the robot to automatically move based on the 3D pose of detected objects

We also plan to test this program with our own YOLO model soon, once more polished.

7/17/2025

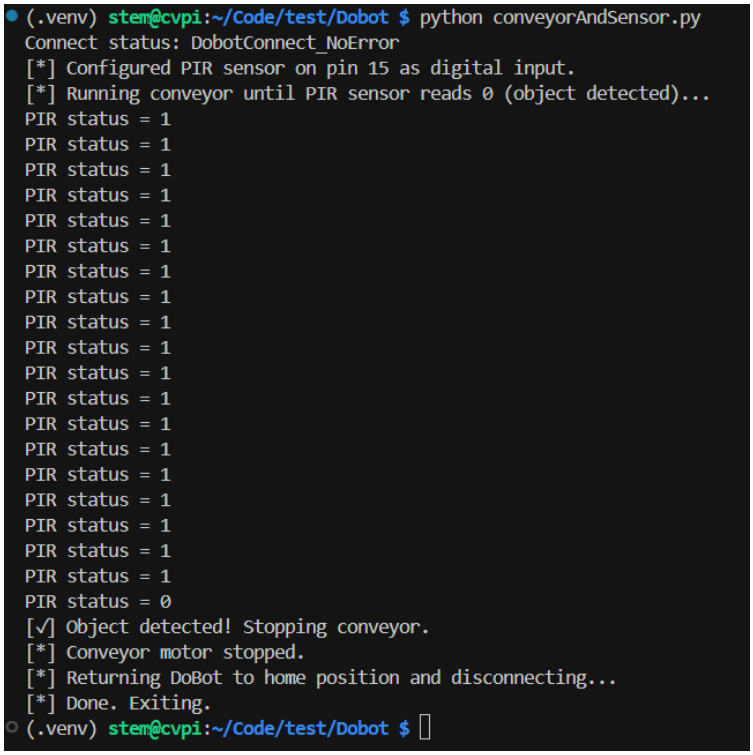
Infrared Sensor Integration and Testing

Integration Summary:

- Dr.Zhou provided updated IR sensor interface files (Python wrapper and C shared library). These were used to update our current DOBOT files in ~/Code/test/Dobot/.
- I verified basic PIR operation using the supplied main.py test script: the terminal prints “1” when idle and “0” when an object is detected.

Conveyor & Sensor Test:

- Developed conveyorAndSensor.py to run the conveyor at a constant speed while polling the PIR sensor.
- On sensor state transition (1 -> 0), the script stops the conveyor, returns the arm to home, disconnects cleanly, and prints “Object Detected!” to the terminal.
- The source Code and a short **demo video** are available here: https://drive.google.com/drive/folders/15ShWcIYSQOxK95Ni9DDTW1X1PMdvl3Vu?usp=sharing



Screenshot: Terminal Output of conveyorAndSensor.py

Next Steps:

- Integrate the PIR-triggered stop logic into the full scanning pipeline to coordinate conveyor run with UR5e depth scanning.
- Once the RealSense RGB stream issue is resolved, combine IR sensor feedback with YOLO detection to automate precise stop positions on the conveyor.
- Use the stable IR sensor framework to ensure reliable handoff timing between the conveyor and robotic arm stages.

7/16/2025

Scanning Script with Object Detection and Depth Alignment

Current Progress:
Im currently working on a script similar to my scanning script I produced a week or two ago. The only difference is, this script uses object detection, and depth alignment.
The purpose of the script is to:

- Ask the user to enter the object to detect.
- Place object(s) on the conveyor with the DOBOT arm.
- Run the conveyor in a separate thread for an infinite amount of time.
- The UR5e with the depth camera will be located at a position in the middle of the conveyor, where we will be using object detection to detect a specific object, rejecting others, during testing this object has been a mouse (Still waiting for our YOLO model).
- Once the object is detected, the conveyor will come to a stop and the UR5e arm will then get the depth, where it will adjust to exactly 10 inches above the object.
- We will then use the Gocator scanner to receive information regarding the object. The 10 inch depth is useful here because that is the most comfortable height to get a good/reliable scan.
- The program will then use all the information gained from the scan and create a datasheet for the object.

Current issue:
The current issue that needs resolved to get this script running deals with the depth camera itself. For some reason today the depth camera stopped working. Me and Nitesh troubleshooted the issue and found that we can stream the depth from the camera but not the RGB, however in the realsense-viewer, we are able to see the RGB stream. We are continuing to troubleshoot this issue and once resolved, the script should nearly be complete. A screenshot of the current error we're receiving can be seen below:

```
(.venv) stem@cvpi:~/Code/test/Object-Detection $ python test_camera.py
Traceback (most recent call last):
  File "/home/stem/Code/test/Object-Detection/test_camera.py", line 10, in <module>
    frames = pipeline.wait_for_frames()
RuntimeError: Frame didn't arrive within 5000
(.venv) stem@cvpi:~/Code/test/Object-Detection $
```

Screenshot: RealSense Camera Error Message

7/17/2025

Update:
Today I tried getting the camera working again using some methods found in the forum below: https://support.realsenseai.com/hc/en-us/community/posts/18310164792851--RuntimeError-Frame-didn-t-arrive-within-5000
I specifically tried using the script provided in the forum to try and get the camera working, I also made and tested the suggested edits to the script that can be seen in the forum. In the end, none of the tests got the RGB stream to work for the Real Sense Camera, and we will continue to troubleshoot this issue.

7/22/2025

Realsense RGB/Depth Refactor
The camera is now working! Nitesh changed up the camera pipeline so that:

- OpenCV now handles the RGB stream
- Librealsense SDK handles depth independently
- Depth values were verified against known distances using the test_camera.py script

```python
# start RealSense depth pipeline
depth_pipeline = rs.pipeline()
depth_cfg = rs.config()
depth_cfg.enable_stream(rs.stream.depth, 640, 480, rs.format.z16, 30)
depth_pipeline.start(depth_cfg)

# open OpenCV capture for RGB
cap = cv2.VideoCapture('/dev/video4', cv2.CAP_V4L2)
cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc(*'YUYV'))
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
cap.set(cv2.CAP_PROP_FPS, 30)
```

Screenshot: Depth and RGB Capture logic.

YOLO Detection Integration
We have wired in ultralytics.YOLO(“yolo11n.pt”) directory on the BGR frames:

- Frames are passed through model.predict(...) in real time
- We compute the box centroid (box_cx, box_cy) and return as soon as the target label is found

```
results = model.predict(source=color_bgr, stream=False, verbose=False)[0]
```

Screenshot: Code Snippet for YOLO detection.

Infrared (PIR) Sensor Stop Logic

After object detection, the conveyor no longer stops immediately but continues until:

1. wait_for_pir() polls the PIR pin bia DobotDllType.GetIODI()
2. Sensor value transitions from 1 -> 0 (object arrives)
3. conveyor.stop() is called, halting the motor and homing the DOBOT arm

```
def wait_for_pir():
    print("[*] Waiting for PIR sensor trigger (object arrival)...")
    dType.SetIOMultiplexing(thread_bot.api, PIR_PIN, 3, 1)
    while True:
        level = dType.GetIODI(thread_bot.api, PIR_PIN)[0]
        print(f"PIR={level}")
        if level == 0:
            print("[✓] PIR sensor triggered!")
            break
        time.sleep(0.05)
```

Screenshot: PIR Sensor Logic.

Depth Alignment & Gocator Scan

As before, once conveyor stops:

1. We sample depth around the detected pixel, discarding measurements under 177.8 mm, and sampling 10 times.
2. Compute average distance, adjusting UR5e Z-axis to exactly 10 inches above the object/conveyor
3. Invoke scan_gocator() to perform the 3D scan

The source code and a short **demo video** demonstrates the full pipeline:
Conveyor Run -> Detection -> Conveyor Run -> IR Stop -> UR5e Alignment -> Gocator scan
https://drive.google.com/drive/folders/1VhVcnLvNZlF8oG_HZhA5_moj7LRDYFSM?usp=sharing

Next Steps

1. Custom YOLO Model: Swap in our trained yolo model (via HailoRT) and verify target detection. We are already in the process of doing this, further testing just needs to be done.

2. Post-Scan UI: After gocator completes, the scan image will be displayed to the user and prompt the user with the options below:

    - Accept Scan - save all scan data
    - Rescan - repeat the Gocator capture, and display the new scan
    - Move Up/Down - nudge the UR5e +/- 10mm in Z based on scan feedback
    - Realign - re-compute depth and re-move to the 10 inch ideal scan height.

    The user will then be able to use any of these options via Voice Control or Buttons.

7/24/2025

Improving Camera FPS and Pipeline Optimization

With the core scanning pipeline working end-to-end (conveyor -> detection -> IR Stop -> Depth Alignment -> Gocator Scan), we refactored the camera/YOLO integration to achieve near-real-time performance (around 30 FPS) and a cleaner, more maintainable codebase. The key changes are summarized below.

Decouple Capture, Detection, and Display

- Multithreading with a frame queue
    - Introduced a queue.Queue(maxsize=1) for RGB frames, a dedicated detection thread, and a display loop in the main thread.
    - This prevents blocking: capture -> enqueue -> immediate return, letting display and detection consume frames independently
- Single Model Load
    - Yolo Model is instantiated once in the detection thread, not per-frame, eliminating repeated initialization overhead.
    - Detection thread calls model.predict(...) on queued frames, signalling completion via a threading.Event.
- Separate depth pipeline
    - Realsense depth pipeline is started once in the display loop. Depth frames are stored in a shared latest_depth_frame under a lock for later scanning, avoiding repeated start/stop.

Optimized OpenCV Settings

- UVC Capture in YUYV mode
    - Configured cv2.VideoCapture with CAP_V4L2 and YUYV four-cc, which yields higher through the Raspberry Pi USB.
- Fixed Resolution and FPS hints

- Set 640x480 @30 FPS explicitly - Camera honors these hints when combined with YUYV

Depth Visualization and Height Map Legend

- Clamped normalization
  - Depth values clamped between DEPTH_DISPLAY_MIN and DEPTH_DISPLAY_MAX (e.g. 100mm to 500mm), then linearly mapped to 0-255
  - Apply cv2.applyColorMap(..., COLORMAP_JET) over this normalized image produced a height map where color corresponds directory to distance
- Inline legend bar
  - Generated a 1D gradient bar beneath the main view, color-mapped in JET, annotated with approximate min, mid, and max depth values (in mm).
  - This provides users with an immediate reference for interpreting colors on the depth stream.

Screenshot: Final “Live View” with side-by-side RGB/depth and color legend.

Code Snippets to Highlight

```
# detection thread
model = YOLO(MODEL_PATH)
while not finish_event.is_set():
    frame = frame_queue.get(timeout=0.1)
    results = model.predict(source=frame, stream=False)[0]
    ...
```

Screenshot: Frame queue and model loop

```
depth_image = np.asanyarray(depth_frame.get_data()).astype(np.float32)
clamped = np.clip(depth_image, MIN, MAX)
norm = ((clamped - MIN)/(MAX - MIN)*255).astype(np.uint8)
depth_colormap = cv2.applyColorMap(norm, cv2.COLORMAP_JET)
...
# build color legend bar
gradient = np.linspace(0,255,width).astype(np.uint8)
legend = cv2.applyColorMap(np.tile(gradient,(bar_h,1)), cv2.COLORMAP_JET)
```

Screenshot: Depth normalization & legend

Demo and Source Code

A link to the source code and a short **demo video** can be seen below:
https://drive.google.com/drive/folders/1LWRIQLPg929YBk_JQ7wR4SvawgZrkuYl?usp=sharing

These optimization raised the camera feed from a sluggish <10 FPS to a consistent 30 FPS, while maintaining full scanning pipeline functionality and user-friendly visualization

7/27/2025 - 8/09/2025

Integrating Custom YOLO into the Scanning Pipeline

Overview:
This chapter documents the final steps in integrating our trained YOLOv11-s model (deployed via HailoRT) into the end-to-end scanning script (detect_test.py), replacing the placeholder model and achieving reliable in-pipeline detection without sacrificing the existing multi-threaded, depth-aligned, Gocator-scanning workflow.

Implementation Highlights

1. Pre-initialize streams and window:
   Before spinning up any threads, we open the RGB and depth pipelines and create the OpenCV window

```
color_cap = cv2.VideoCapture('/dev/video4', cv2.CAP_V4L2)
color_cap.set(cv2.CAP_PROP_FRAME_WIDTH,  640)
color_cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
depth_pipe = rs.pipeline()
cfg = rs.config()
cfg.enable_stream(rs.stream.depth, 640, 480, rs.format.z16, 30
depth_pipe.start(cfg)

cv2.namedWindow("Live View", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Live View", 1200, 620)
```

Screenshot: Code snippet showing pre-initialization.

2. Single-load Hailo model with threshold:
   We load the model once (not per-frame) and set confidence_threshold = 0.5. This change passes detections with confidence levels above 0.5 (50%).

```
hailo_model = dg.load_model(
    MODEL_NAME, inference_host_address=INFERENCE_HOST,
    zoo_url=ZOO_URL, token='', device_type=DEVICE_TYPE
)
hailo_model.confidence_threshold = CONFIDENCE_THRESH  #
```

Screenshot: Code Snipper showing hailo model initalization.

3. Full pipeline orchestration:

   - Conveyor starts immediately after the viewing window appears.
   - Detection updates detected_box as soon as YOLO finds the target label
   - PIR stop halts the conveyor, homes the Dobot, and joins the thread
   - Depth alignment samples 10 depth frames around the detection pixel, averages valid readings, and adjusts the UR5e's Z to 10 inches above the object.
   - Gocator scan invokes the same C-bindings, wrapped in a try/except to disregard Open3D errors and guarantee finish_event.set()

Results

- Detection performance now matches the standalone demo: gears are detected as they pass on the conveyor without having to pause under the camera.
- Pipeline stability improved: no more segmentation faults or hanging threads after scans; all stages (conveyor, detection, PIR, UR5e moves, scan) execute in sequence.
- Window load time is the only issue seen still. The initial load of the viewing window takes approximately 20 seconds, but once loaded the FPS is high (around 30), and the view remains until the full cycle is complete.

A short **demo video** along with the source code be seen via the link below:
https://www.dropbox.com/scl/fo/eua1d0ti9xd9n0xa7jen0/AFhB57jUjFBC5ns_0Nimoe0?rlkey=rk8x98m3gsfqxdvtnlzybw2at&st=8tu32s6z&dl=0

Next Steps:

1. Generating an inspection report: After the scan completes, the dimensions of the object will be measured and compared to the “ideal” measurements to see if the object passes or fails the inspection. These measured values, a picture, and the pass / reject decision will all be included in the document.
2. Implementing the post-scan UI mentioned previously: most likely removing the up/down options since that will interfere with the Gocator’s alignment with the conveyor

7/31/2025

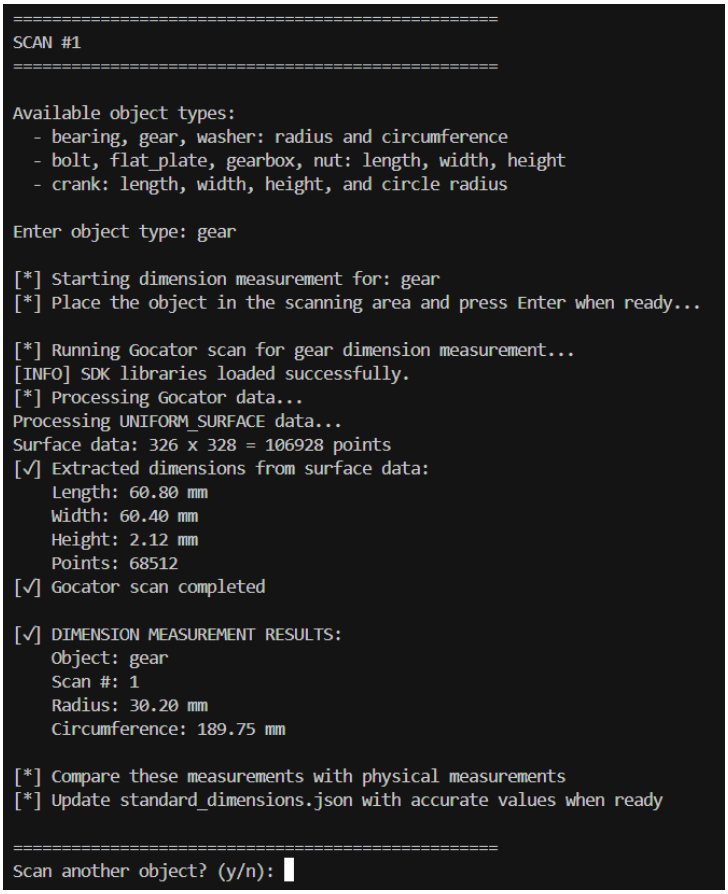
Dimension Tracking & Inspection Report Generation

dimension_tracker.py

The objective of today’s work was to begin automating the inspection process by generating a complete inspection report with three key components:

1. A 2D scan image of the object.
2. The object’s scanned dimensions.
3. A pass/fail status compared to predefined standards

To accomplish this, I first created a program titled dimension_tracker.py, which allows for continuous scanning of objects using the Gocator sensor. This script prompts the user for the object type, performs a 3D scan, and returns key measurements (e.g., length, width, height, radius, and circumference depending on the object.)
The robot arm moves to a defined scanning pose, then captures and processes data from the Gocator, providing results in the terminal after each scan. A summary is printed at the end of each session, reminding the user to update the object standards accordingly. An image of this programs output can be seen below:



Screenshot: Terminal output of dimension_tracker.py.

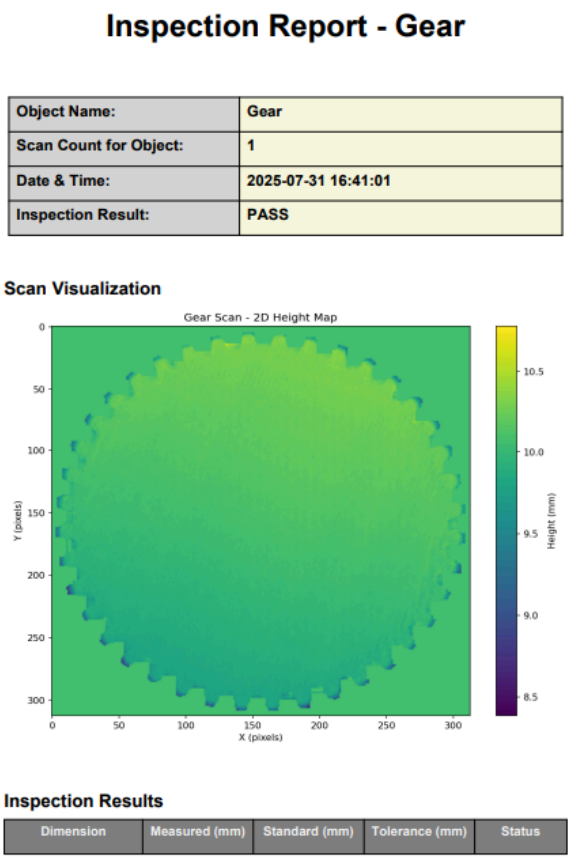
Using this program, I scanned multiple objects and compared the Gocator's output with physical measurements to derive accurate baseline dimensions for each object. The finalized data was saved into a file titled **standard_dimensions.json**, which contains entries for object types such as **gear, bearing, bolt, washer, flat_plate, crank, gearbox, and nut**, along with a configurable **tolerance value.**

report.py

With that data established, I then built report.py - a script designed to generate inspection reports in PDF format. The workflow consist of:

- Prompting the user to identify the object type.
- Moving the UR5e arm to scanning position.
- Capturing a Gocator snapshot and image.
- Comparing measured values to the object's standards.
- Classifying each dimension as pass/fail based on tolerance
- Generating a PDF report with the object's name, scan number, scan date/time, scan image, and all dimension results.

The image is rendered as a 2D height map using Matplotlib and embedded in the report, while tabular summaries of measurements are styled with color-coded pass/fail labels using ReportLab.



Inspection Report - Gear

Object Name:	**Gear**
Scan Count for Object:	**1**
Date & Time:	**2025-07-31 16:41:01**
Inspection Result:	**PASS**

Scan Visualization

Inspection Results

Dimension	Measured (mm)	Standard (mm)	Tolerance (mm)	Status

Screenshot: A piece of the generated PDF inspection report.

Initial tests show the program works as intended. All generated reports are saved to the inspection/documents/ directory and named based on object type and scan count (e.g., gear1Scan.pdf, gear2Scan.pdf, etc.)

The source code for both of these programs, the .json file containing the standard measurements, and the tested output PDF files can be seen via the link below: https://drive.google.com/drive/folders/1wDQcktmbs5NXcKTwBDOVK5VwAxQeSctT?usp=sharing

- gear1Scan.pdf displays a PASS when scanning a gear that meets specifications
- gear2Scan.pdf correctly identifies a larger gear and flags it as FAIL

The next step is to integrate this inspection and reporting tool into the full robotics pipeline, potentially with a post-scan UI for object review and operator feedback.

8/04/2025

Post-Scan UI Development

Overview
To improve operator interaction at the end of our scanning pipeline, we designed and implemented a lightweight Tkinter-based “post-scan” review interface. After the UR5e completes its depth alignment and the Gocator capture, this UI presents the user with:

- A rendered 2D height-map scan image.
- A table of measured dimensions (pass/fail color-coding).
- Three Action buttons (will also implement voice control for these in full pipeline):
    - “Rescan” - if the user notices some small errors in the scan, say the height map is a little off, they can select this option to rerun the scan, they will then be prompted again with the new image and measurements gathered, with the same options.
    - “Realign” - if the user sees an apparent error with the height of the scan, the can select this option to rerun the 10 inch alignment method, which takes 10 samples of the depth and adjusts the UR5e’s z axis to be exactly 10 inches from the surface beneath it (ideal scan height).
    - “Accept Scan” - if the user believes the image of the scan and the measurements presented are accurate, the user can select this option thus generating the inspection report.

Implementation Details

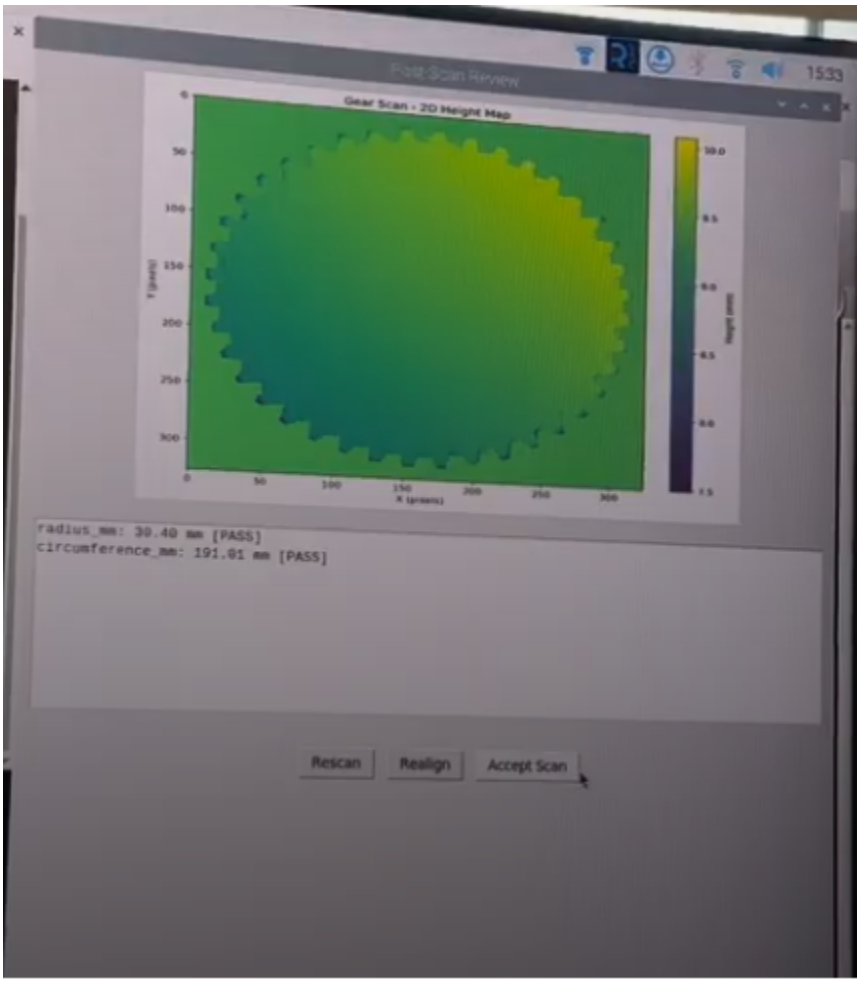
1. Script Location & Imports
    - Path: ~/Code/test/Object-Detection/post-scan-ui/post_scan_ui.py
    - Reuses our existing report.py utilities (run_gocator_scan(), calculate_object_measurements(), perform_inspection(), generate_pdf_report(), etc.)
    - Leverages tkinter, PIL (for image display), and rtde_control for UR5e moves
2. Workflow Thread
    - On startup, window remains hidden (self.withdraw()) while a background thread (run_sequence) runs:
        - Prompt for object label

- Perform the 10 inch alignment (10 samples of depth at the center pixel -> adjust UR5e Z)
- Move to scan pose at that height and run run_gocator_scan()
- Compute object-specific measurements and inspection results
- Once scan completes, self.deiconify() displays the UI

3. UI Layout

- Image Panel: upper portion, displays the saved scan PNG (resized to 600 x 400)
- Measurements text box: below the image, lists each dimension with its measured value and PASS/FAIL status
- Button row: at the bottom, three Tkinter Button widgets.

An image showing how the Post Scan UI looks can be seen below:



Screenshot: An image of the Post-Scan UI.

Testing and Results

- Rescan reliably produces a new height-map image without re-running alignment
- Realign re-positions UR5e Z by fresh depth sampling; user then manually clicks Rescan or Accept
- Accept Scan generates a properly styled PDF (as in report.py) and closes the UI
- Operator feedback: intuitive button layout, fast scan-to-UI turnaround (< 3 seconds after scan), no threading conflicts.

A link containing the source script along with a short **demo video** can be seen below:
https://drive.google.com/drive/folders/178lVCBjQb9AQwhn-5qKDKcBgdIDQ4IBk?usp=sharing

This post-scan UI bridges automated data acquisition with human review, enabling operators to validate or repeat scans before generating final inspection reports. Future enhancements may add conveyor or arm override controls directly in this interface.